

# Podpora jazyka Monkey C v prostředí VS Code

Monkey C Language Support in VS Code

Jakub Pšenčík

Bakalářská práce

Vedoucí práce: Ing. Jan Janoušek

Ostrava, 2021

## **Abstrakt**

V této bakalářské práci se budu zabývat vývojem rozšíření pro Visual Studio Code, jenž bude poskytovat podporu pro jazyk Monkey C. V teoretické části dojde k představení prostředí Visual Studio Code a nahlédnutí do problematiky vývoje rozšíření v tomto prostředí. Dále bude představen také jazyk Monkey C. V praktické části práce bude popsán nástroj ANTLR, který je schopen generovat vlastní překladač jazyka pomocí bezkontextové gramatiky, parsováním kódu a jeho syntaktickou analýzou. Dále bude v praktické části rozebrán návrh a implementace rozšíření, společně s popisem jednotlivých jeho částí. Závěrem bude výsledné rozšíření testováno a výsledky zhodnoceny.

## **Klíčová slova**

bakalářská práce, rozšíření, Monkey C, Typescript, parser

## **Abstract**

In this bachelor thesis I will deal with the development of extension for Visual Sstudio Code, which will provide full support for Monkey C language. In thoretical part will be introduced Visual Studio Code environment and insight into the development of extensions in this environment. The practical part of the thesis will describe the ANTLR tool, which is able to generate its own language compiler using context-free grammar, code parsing and its syntactic analysis. Furthermore, the practical part will discuss the design and implementation of the extension, along with a description of its individual parts. Finally, the resulting extension will be tested and the results evaluated.

## **Keywords**

bachelor thesis, extension, Monkey C, Typescript, parser

## **Poděkování**

Rád bych na tomto místě poděkoval svému vedoucímu práce Ing. Janu Janouškovi za odborné a metodické vedení, ochotu a pomoc v průběhu vypracovávání práce.

# Obsah

<b>Seznam použitých symbolů a zkratek</b>	<b>6</b>
<b>Seznam obrázků</b>	<b>7</b>
<b>Seznam tabulek</b>	<b>8</b>
<b>1 Úvod</b>	<b>9</b>
<b>2 Jazyk Monkey C</b>	<b>10</b>
<b>3 Problematika vývoje rozšíření pro VS Code</b>	<b>13</b>
3.1 Visual Studio Code . . . . .	13
3.2 Typescript . . . . .	14
3.3 Komponenty potřebné pro tvorbu rozšíření . . . . .	14
3.4 ANTLR - Nástroj pro generování překladače . . . . .	14
<b>4 Syntaktická a sémantická analýza kódu</b>	<b>18</b>
4.1 Parser a Lexer . . . . .	18
4.2 Syntaktický strom . . . . .	20
4.3 ANTLR Listener a callback funkce . . . . .	22
4.4 Sémantická analýza . . . . .	23
<b>5 Návrh a implementace rozšíření</b>	<b>25</b>
5.1 Třída extension.ts . . . . .	25
5.2 Třída documentHandler.ts . . . . .	29
5.3 Třída Listener.ts . . . . .	30
5.4 Error Listener . . . . .	31
5.5 Modul Toybox . . . . .	31
5.6 Obarvení kódu . . . . .	32
5.7 Automatické doplňování a našeptávání kódu . . . . .	32
5.8 Popis funkcí a proměnných Toyboxu . . . . .	33

5.9	Nedostatky rozšíření . . . . .	34
<b>6</b>	<b>Testování výsledného řešení</b>	<b>36</b>
<b>7</b>	<b>Závěr</b>	<b>38</b>
	<b>Literatura</b>	<b>40</b>
	<b>Přílohy</b>	<b>41</b>
<b>A</b>	<b>Testovací zdrojové kódy</b>	<b>42</b>

# Seznam použitých zkratek a symbolů

ANTLR	– ANother Tool for Language Recognition
AST	– Abstract syntax tree
VS Code	– Visual Studio Code
API	– Application Programming Interface

# Seznam obrázků

2.1	ukázka jednoduchého fragmentu kódu v MonkeyC . . . . .	11
2.2	rozšíření při pokusu předat funkci, jako parametr, detekuje chybu. . . . .	12
3.1	ukázka hlavičky gramatiky MonkeyC.g4 pro popis jazyka . . . . .	15
3.2	potřebné soubory vygenerované nástrojem ANTLR . . . . .	16
3.3	"ParseTreeInspector"- vizuální podoba syntaktického stromu . . . . .	17
4.1	přiřazení hodnoty 1 proměnné input . . . . .	19
4.2	Ukázka, jak Language recognizer zpracovává vstupní sekvenci znaků . . . . .	19
4.3	funkce vygenerované nástrojem ANTLR . . . . .	23
4.4	ukázka tabulky symbolů [14] . . . . .	23
5.1	popis konstruktoru třídy CompletionItem z VS Code API . . . . .	27
5.2	rozšíření našeptává třídy z importovaných modulů. . . . .	28
5.3	rozšíření našeptává datové typy z modulu Toybox.Activity. . . . .	29
5.4	vytvoření instancí parseru a lexeru a následné parsování soboru. . . . .	30
5.5	Monkey C kód před a po přidání obarvení syntaxe . . . . .	32
5.6	ukázka automatického našeptávání kódu na proměnné typy string . . . . .	34
5.7	komentář nad funkcí obsahující stručný popis, parametry funkce a návratový typ . .	34
5.8	popis funkce registerSensorDataListener() z modulu Toybox.Sensor. . . . .	35
5.9	nedostatek rozšíření . . . . .	35
5.10	chybová hláška z error listeneru . . . . .	35
6.1	rozšíření našeptává konstanty obsahující písmeno D. . . . .	37

# Seznam tabulek

3.1	počet výsledků po vyhledání daného jazyka na marketplace [2]	13
-----	--	----



# Kapitola 1

## Úvod

V dnešní době, kdy jsme obklopeni spoustou moderních technologií, existující programovací jazyky se stále vyvíjejí kupředu a nové postupně vznikají, existuje nespočet nástrojů, rozšíření, vývojových prostředí, které práci a vývoj v těchto jazycích dokážou v mnoha ohledech usnadnit. Jazyk Monkey C, jenž bude středobodem této práce, zatím nedisponuje tak široce obsáhlou komunitou, jako mají např. v dnešní době velmi populární Javascript, Python, C Sharp, Ruby, atd...

Typickým příkladem společnosti, která se zabývá právě vývojem softwarů pro programátory či vývojáře, je česká JetBrains s.r.o. Ti mají na kontě již mnoho produktů usnadňujících programování, např. ReSharper (.NET), IntelliJ IDEA (Java, Groovy, atd...) či PyCharm (Python). Ovšem na podporu Monkey C zatím žádné softwarové řešení v JetBrains do světa nepustili. [1]

Na oficiálním webu, kde Microsoft nabízí nejrozšířenější rozšíření [2], je sice možné nalézt několik nástrojů, které s vývojem v Monkey C pomáhají. Cílem této práce je však tvorba rozšíření, které poskytne uživateli co největší podporu a hlavně, aby byla postavena na vlastním řešení. Následujících kapitoly tedy budou věnovány tvorbě a vývoji rozšíření pro vývojové prostředí Visual Studio Code, které poskytne plnou podporu při vývoji aplikací v Monkey C. V kapitole 2 dojde k představení samotného jazyka Monkey C a operačního systému Connect IQ, na kterém Monkey C běží. V kapitole 3 jsou popsány veškeré nástroje a komponenty, které budou při tvorbě rozšíření použity. Pro tvorbu rozšíření bude využit jazyk Typescript, dále pak nástroj ANTLR, který je schopen vygenerovat překladač jazyka. Toho je schopen dosáhnout za pomoci jeho popisu obsaženého v bezkontextové gramatice, jenž byla pro tuto práci poskytnuta, a tudíž její vytváření nebylo součástí práce. Kapitola 4 se popisuje analýzu kódu jak z pohledu syntaxe, tak sémantiky. V kapitole 5 se práce zabývá již samotným návrhem a implementací rozšíření. Objeví se zde témata, jako parsování kódu, automatické doplňování a našptávání kódu, atd... Kapitola 6 je poté věnována testování rozšíření.

## Kapitola 2

# Jazyk Monkey C

Monkey C [3] je objektově orientovaný jazyk. Byl navržen americkou společností Garmin Ltd. [4] a jeho hlavním účelem je snadný vývoj Connect IQ aplikací pro nositelná zařízení (nejvhodnějším příkladem jsou zde chytré hodinky). Jedná se o dynamický programovací jazyk, podobně jako Java, PHP či Ruby. Z těchto uvedených jazyků také Monkey C vychází. Cílem Monkey C je zjednodušit proces vytváření samotné aplikace a umožnit tak vývojářům více se soustředit na zákazníka a méně na omezení zdrojů. Využívá tzv. "reference counting" k automatickému čištění paměti, což vývojáře osvobozuje od manuální správy paměti (např. jako v jazyce C/C++).

Všechny aplikace, naprogramované v Monkey C, běží na operačním systému (vývojářské platformě) Connect IQ. [5] Connect IQ umožňuje jak jednotlivcům, tak velkým společnostem vyvíjet nespočet různých aplikací a rozšíření pro Garmin zařízení. Můžeme je tedy zařadit mezi takové platformy, jako jsou Android či iOS. S pomocí Connect IQ je tedy možné vyvíjet např.:

1. **aplikace** - jedná se o plně funkční aplikace běžící na hodinkách. Může se jednat např. o hudební aplikaci, která synchronizuje obsah s mobilní aplikací na zařízení uživatele (např. Spotify, iTunes, atd...)
2. **ciferníky** (Watch Faces) - ciferník si lze představit, jako domovskou obrazovku na telefonu, které uživateli dokáže zobrazit celou řadu informací (záleží samozřejmě na preferencích konkrétního jedince). Ciferník je na hodinkách aktualizován každých 60 vteřin a běží nepřetržitě v režimu nízké spotřeby.
3. **widgety** - opět se jedná o komponentu, která je běžně využívána na mobilních zařízeních. Widget dokáže, za pomoci dat z hodinek nebo připojeného telefonu, zobrazovat nejrozličnější informace od počasí, přes puls uživatele až po notifikace příchozích hovorů.

Na obrázku 2.1 je možnost vidět jednoduchý fragment kódu v Monkey C. Z obrázku je patrné, že Monkey C, stejně jako většina dnešních programovacích jazyků, podporuje operace, jako dědičnost, určení rozsahu jmenného prostoru pomocí klíčového slova `using`, a mnoho dalších. "Stejně jako je

```

using Toybox.Application as App;
using Toybox.System;

class MyProjectApp extends App.AppBase {

    // onStart() is called on application start up
    function onStart(state) {
    }

    // onStop() is called when your application is exiting
    function onStop(state) {
    }

    // Return the initial view of your application here
    function getInitialView() {
    return [ new MyProjectView() ];
    }
}

```

Obrázek 2.1: ukázka jednoduchého fragmentu kódu v MonkeyC

tomu v programovacím jazyce Java, Monkey C kompiluje zdrojové soubory do byte kódu, který je následně interpretován virtuálním strojem Monkey Brains. Virtuální stroj Monkey Brains poté komunikuje s dalšími dostupnými API.[6] Tyto API slouží např. pro práci s polohou zařízení, komunikací se senzory detekující nejrůznější informace (teplotu ovzduší, tlak vzduchu, puls, atd...).

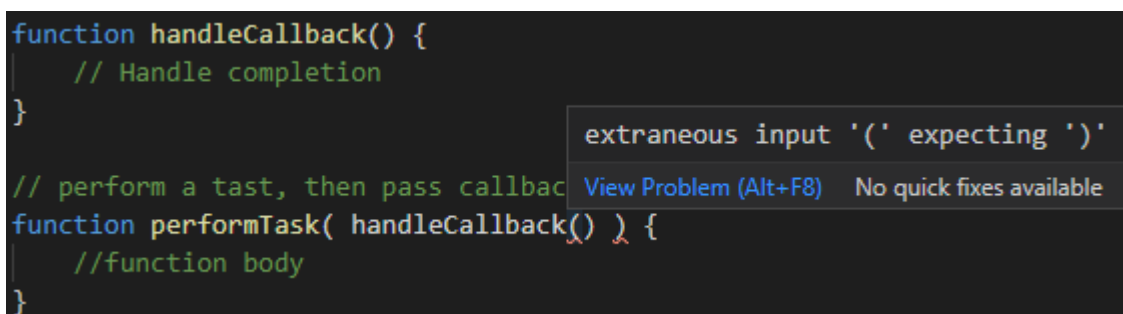
### 2.0.1 Monkey C a ostatní jazyky

Stejně jako italština a španělština vznikly z latiny, Monkey C čerpá spoustu věcí převážně z jiných moderních jazyků. C, Java, JavaScript, Python, Lua, Ruby a PHP, všechny tyto jazyky ovlivnily výslednou podobu Monkey C. [3]

Mezi Monkey C a výše uvedenými jazyky jsou přesto jisté rozdíly, které budou popsány v následujících řádcích.

1. **Java** - Stejně, jako Java, je Monkey C kompilován do byte kódu, který je následně interpretován virtuálním strojem. Jak už bylo zmíněno výše, virtuální stroj pro Monkey C nese název Monkey Brains. Další společná vlastnost s Javou je ta, že k alokaci objektů dochází na haldě (heap). V neposledí řadě stojí za zmínku, že virtuální stroj má na starosti čištění paměti, přičemž v Javě je toto realizováno prostřednictvím tzv. "garbage collectoru" a v Monkey C pomocí "reference countingu".

2. **JavaScript** - Hlavním rozdílem mezi JavaScriptem a Monkey C spočívá v tom, že funkce v Monkey C nedisponují vlastností "funkce první třídy" (first-class function). "To znamená, že jazyk podporuje předávání funkcí jako argumentů jiným funkcím, jejich vrácení jako hodnoty z jiných funkcí a jejich přiřazení k proměnným nebo jejich uložení v datových strukturách." [7] Na obrázku 2.2 lze vidět, že při pokusu předat funkci, jako parametr jinou funkci, rozšíření detekuje operaci, která je v rozporu s Monkey C gramatikou, a tím pádem oznámí uživateli chybu.
3. **Python** - Objekty v Pythonu jsou reprezentovány, jako hashovací tabulky, přičemž funkce a proměnné lze objektům přiřazovat za běhu programu. Objekty v Monkey C jsou kompilovány ještě před spuštěním programu, a tím pádem nemohou být modifikovány za běhu. Všechny proměnné, předtím než je použijeme např. ve funkci, třídní instanci či rodičovském modulu, musí být deklarovány.



```
function handleCallback() {  
    // Handle completion  
}  
  
// perform a task, then pass callback  
function performTask( handleCallback() ) {  
    //function body  
}
```

The screenshot shows a code editor with a TypeScript error. The code defines a function `handleCallback()` and then calls `performTask( handleCallback() )`. A red squiggly line under the opening parenthesis of `handleCallback()` indicates an error. A tooltip displays the message: "extraneous input '(' expecting ')'" with a link to "View Problem (Alt+F8)" and the text "No quick fixes available".

Obrázek 2.2: rozšíření při pokusu předat funkci, jako parametr, detekuje chybu.

## Kapitola 3

# Problematika vývoje rozšíření pro VS Code

Jak už bylo zmíněno v úvodní kapitole, na oficiálním webu s rozšířeními pro VS Code [2] najdeme spousty aplikací, které vývojářům pomáhají např. s "debuggováním" kódu, analýzou dat, strojovým učením (machine learning), atd... Rozšíření pro Monkey C však nejsou zastoupena v tak hojném počtu, jako ostatní jazyky, což lze vidět v tabulce níže 3.1. V této kapitole budou popsány komponenty, které jsou stěžejní pro vytvoření překladače jazyka, parseru jazyka a následně rozšíření samotného.

### 3.1 Visual Studio Code

Visual Studio Code [8] je editor zdrojového kódu vytvořený společností Microsoft pro operační Windows, Linux a MacOS, který podporuje stovky druhů jazyků. Je naprogramován v jazycích JavaScript a Typescript a nabízí spoustu užitečných funkcí, mezi které patří podpora ladění kódu, zvýrazňování syntaxe, automatické doplňování a našeptávání kódu, odsazování textu, intuitivní klávesové zkratky, které usnadňují navigaci v kódu, atd... Cílem práce je integrovat většinu těchto funkcí a možností do finálního rozšíření.

Tabulka 3.1: počet výsledků po vyhledání daného jazyka na marketplace [2]

název jazyka	počet výsledků
C Sharp	169
Java	2521
Python	454
R	6350
Monkey C	2

## 3.2 Typescript

TypeScript [9] je open-source programovací jazyk vyvinutý společností Microsoft. Jedná se o nádstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektově orientovaného programování jako jsou třídy, moduly a další. Samotný kód psaný v jazyce TypeScript se kompiluje do jazyka JavaScript. Jelikož je tento jazyk nádstavbou nad JavaScriptem, je každý JavaScript kód automaticky validním TypeScript kódem.

S programovacím jazykem Typescript jsem na začátku této práce mnohem menší zkušenosti, než např. s programovacími jazyky C Sharp nebo JavaScript. V průběhu studia jsem se však setkal s JavaScriptem při vývoji internetových aplikací. A jelikož je Typescript pouze nádstavba JavaScriptu a objektově orientované programování mě bylo dobře známé z jazyka C Sharp, nebylo obtížné se veškeré chybějící potřebné znalosti rychle doučit.

## 3.3 Komponenty potřebné pro tvorbu rozšíření

Předtím, než bude možné začít pracovat na samotném rozšíření, je potřeba obstarat několik důležitých nástrojů a komponent, jenž jsou klíčové při vývoji. Nejdůležitějším nástrojem pro vývoj rozšíření je však ANTLR, který je popsán v sekci 3.4.

### 3.3.1 Gramatika

Jako první je potřeba definovat popis jazyka Monkey C. V tomto případě je jazyk Monkey C popsán prostřednictvím bezkontextové gramatiky, která formálně definuje syntax (pravidla) jazyka. Jedná se, ve své podstatě, o soubor pravidel, kde každé pravidlo reprezentuje určitou strukturu či frázi jazyka. Z tohoto formálního popisu je ANTLR schopen vygenerovat parser daného jazyka, který dokáže automaticky sestavit datovou strukturu, která se nazývá buďto "parse tree" nebo "syntax tree", což v češtině znamená **syntaktický strom**. Tento syntaktický strom poté reprezentuje, jak přesně je gramatika schopna rozpoznávat vstupní data (např. fragment kódu).

Jelikož je v této práci použit ANTLR verze 4, je klíčové, aby název souboru obsahující gramatiku končil příponou .g4, v tomto případě soubor nese název **MonkeyC.g4**.

Na obrázku 3.1 lze vidět prvních pár řádků Monkey C gramatiky. Gramatika obsahuje spoustu známých klíčových slov, nebo-li tokenů, např. "CLASS", "FUNCTION", "USING", atd...

## 3.4 ANTLR - Nástroj pro generování překladače

ANTLR nebo také ANother Tool for Language Recognition (jiný nástroj pro rozpoznávání jazyka) [10] je výkonný nástroj pro generování syntaktických analyzátorů, tzv. "parserů". Tento parser je poté schopen číst, zpracovávat, spouštět nebo překládat strukturované textové či binární soubory.

Používá se především k vytváření nových jazyků, nástrojů nebo "frameworků". Využívá bezkontextový jazyk typu LL, což je syntaktický analyzátor typu "shora-dolů" pro bezkontextové gramatiky. Analyzuje vstup zleva (Left) doprava a konstruuje nejlevější derivaci (Leftmost) věty. Gramatiky, které jsou takto analyzovatelné, se nazývají LL gramatiky.

Pro vygenerování Monkey C parseru je tedy potřeba nainstalovat ANTLRv4 (ANTLR verze 4), který je možné získat na oficiálním webu [10]. Stačí tedy stáhnout aktuální ANTLR jar, což je momentálně "antlr-4.8-complete.jar". Soubor je zakončen příponou **.jar**, z toho vyplývá, že je ANTLR napsán jazyce Java. K úspěšnému spuštění ANTLR nástroje je vyžadována verze Javy 1.6 a vyšší.

### 3.4.1 TestRig

ANTLR poskytuje flexibilní testovací nástroj umístěný v runtime knihovně s názvem TestRig. TestRig dokáže poskytnout spoustu informací o tom, jak "recognizéry" (parser a lexer) zpracovávají InputStream ze vstupního souboru. TestRig je spouštěn z příkazového řádku pomocí aliasu *grun*. Nabízí spoustu možností, jak zobrazit vygenerovaný syntaktický strom:

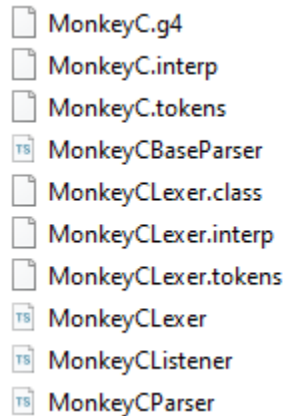
1. **-tree**: zobrazí syntaktický strom jako soubor do sebe zanořených pravidel gramatiky
2. **-tokens**: výstupem jsou tokeny, které parser vygeneroval ze vstupních dat
3. **-gui**: zobrazí syntaktický strom vizuálně v programu ParseTreeInspector, který je součástí ANTLR. Příklad zobrazení jednoduchého vstupu bez detekovaných chyb lze vidět na obrázku

```
grammar MonkeyC;

options {
    superClass=MonkeyCBaseParser;
}
@header {import { MonkeyCBaseParser } from "../MonkeyCBaseParser";}

DOT : '.';
SEMI : ';';
QUES : '?';
COLON : ':';
CLASS : 'class';
FUNCTION : 'function';
RETURN : 'return';
NEW : 'new';
VAR : 'var';
CONST : 'const';
MODULE : 'module';
USING : 'using';
AS : 'as';
ENUM : 'enum';
EXTENDS : 'extends';
```

Obrázek 3.1: ukázka hlavičky gramatiky MonkeyC.g4 pro popis jazyka

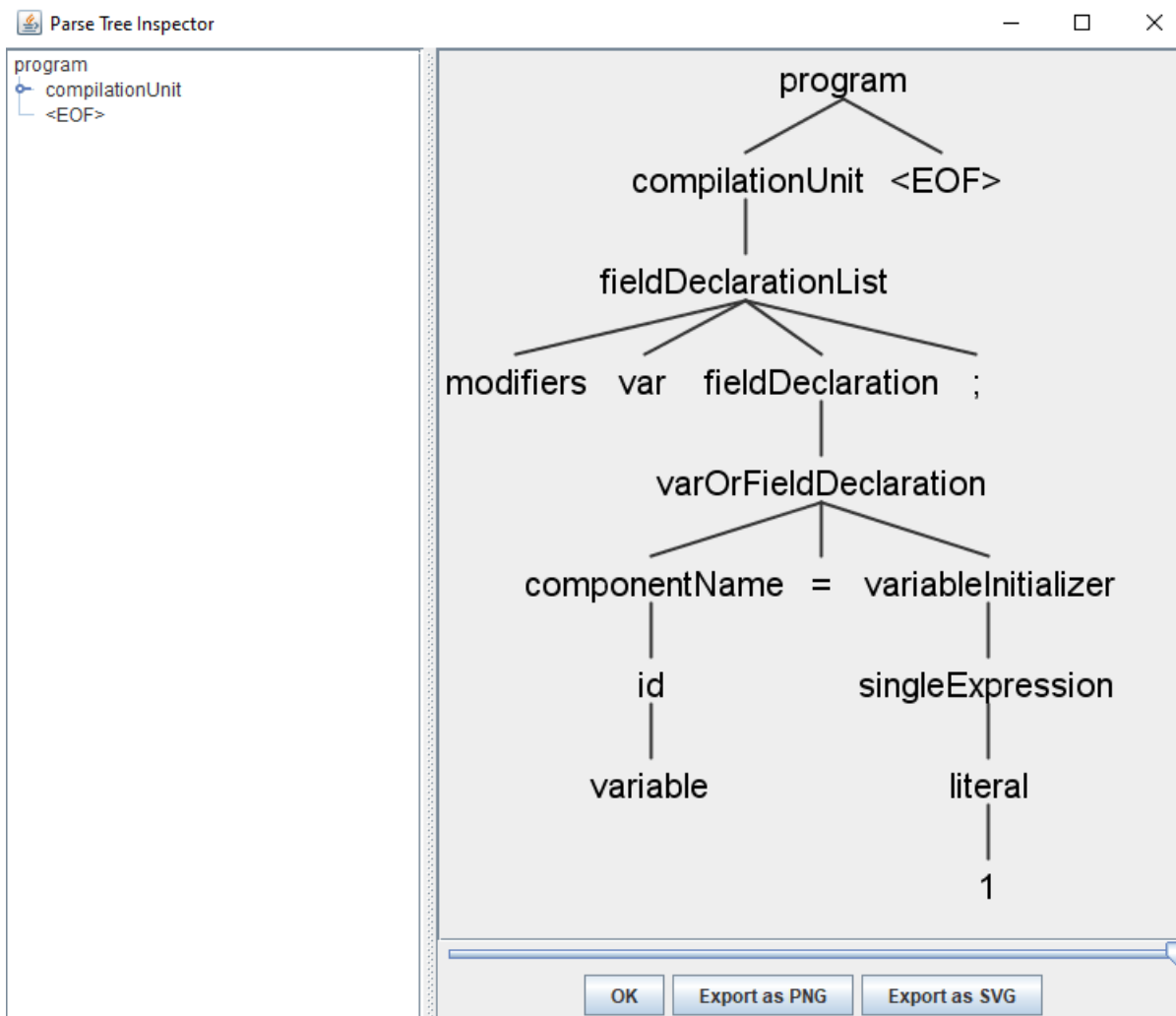


Obrázek 3.2: potřebné soubory vygenerované nástrojem ANTLR

3.3. Dále lze na obrázku vidět, jak jsou mezi sebou jednotlivé části stromu, tedy uzly, navzájem propojeny. Strom začíná kořenem, jenž je v gramatice pojmenován, jako "program". Takto je kořen pojmenován při každém parsování.

Podle mého názoru se jedná o užitečný nástroj, který uživateli poskytuje přehled o tom, zda parser rozpoznal vstupní data správně, případně kde parser detekovat rozpory z gramatikou. Při vývoji a testování rozšíření byl tento nástroj velmi kvalitním pomocníkem. Možností, které pro testování gramatik a parserů TestRig nabízí, je samozřejmě více, ale pro účely této práce byly použity hlavně tři výše uvedené.





Obrázek 3.3: "ParseTreeInspector"- vizuální podoba syntaktického stromu

## Kapitola 4

# Syntaktická a sémantická analýza kódu

Abychom mohli implementovat náš jazyk (MonkeyC), je potřeba vytvořit aplikaci, která je schopna číst "věty" na vstupu a odpovídajícím způsobem rozpoznává klíčová slova či fráze naší gramatiky. (Obecně je jazyk složením platných vět, kdy věta se skládá z frází a fráze se skládá ze symbolů slovní zásoby).

Obecně lze říci, pokud nějaká aplikace vykonává (interpretuje) zápis jiného programu v jeho zdrojovém kódu ve zvoleném programovacím jazyce (v našem případě Typescript), že se jedná o tzv. interpret [11]. Jako příklady je možné uvést kalkulačku, aplikace pro čtení konfiguračních souborů, Python interprety, atd... Pokud, na druhou stranu, dochází k převodu "věty" z jednoho jazyka do druhého (např. z Javy do 'C Sharp'), nazývá se taková aplikace překladačem.

Úkolem překladače či interpreta je tedy rozpoznat platné vstupy, tedy věty, fráze, subfráze, klíčová slova, atd... Přičemž rozpoznáním platného vstupu je myšleno identifikovat jednotlivé fráze a rozlišit je od jiných.

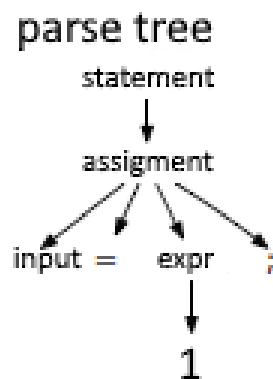
Jako příklad použijeme rozpoznání vstupu `input = 1`; s validní MonkeyC syntaxí. Syntaktický strom z těchto vstupních dat je možné vidět na obrázku 4.1. Z tohoto vstupu je patrné, že "input" je cíl přiřazení a "1" představuje hodnotu, která se má přiřadit. Stejně jako jsme my schopni rozlišit v našem jazyce sloveso od podstatného jména, je naše aplikace schopna rozlišit toto přiřazení od např. importování knihovny pomocí klíčového slova "using".

Programy, které jsou schopny rozpoznávat konkrétní jazyk, se nazývají parsery nebo syntaktické analyzátory, přičemž syntax odkazuje na pravidla obsažená v popisu jazyka.

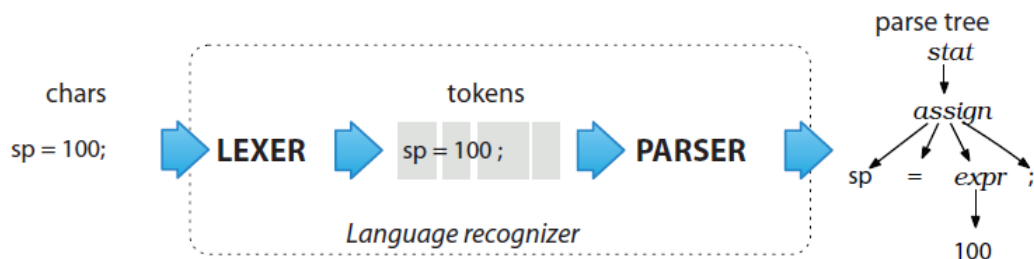
### 4.1 Parser a Lexer

K vytvoření parseru a lexeru je potřeba spustit ANLTR nástroj, který na základě popisu jazyka (gramatiky) tyto soubory vygeneruje. S jejich pomocí je poté možné generovat syntaktické stromy pro vstupní data. Zde platí, že pokud jsou vstupní data validní (nejsou v rozporu s pravidly gra-

matiky), tak je vygenerovaný parser vždy rozpozná správně neohledě na složitost gramatiky. Proces, při kterém dochází ke skládání znaků ze vstupních dat do slov či symbolů (tokenů) nese název **lexikální analýza** nebo **tokenizace**. Program, který provádí tuto tokenizaci se nazývá právě **lexer**. Obecně lze říci, že lexer (nebo také tokenizér) "rozdělí" text na vstupu (Monkey C kód) na tokeny. [12]. Dále lexer dokáže vytvořené tokeny seskupovat do tokenových tříd, nebo typů, např. IDENTIFIER (identifikátor), INTLITERAL (celočíslná proměnná), DOUBLELITERAL (proměnná typu double), atd... V momentě, kdy lexer rozdělil vstupní data na jednotlivé tokeny, jsou tyto tokeny předány parseru. Ten poté "shora dolů" prochází text na vstupu a porovnává jednotlivé řádky s pravidly obsažené v gramatice [12]. Na obrázku 4.2 je možné vidět, jak parser zpracovává vstup `sp = 100;`. Tokeny, které lexer z těchto dat rozpozná, jsou [SP, =, 100, ;, <EOF>], kde SP je název proměnné, 100 hodnota, která je proměnné přiřazena, jako ve většině programovacích jazyků musí být příkaz ukončen středníkem, <EOF>, který se vyskytuje na konci každého validního vstupu, detekuje konec souboru (EOF - End Of File).



Obrázek 4.1: přiřazení hodnoty 1 proměnné input



Obrázek 4.2: Ukázka, jak Language recognizer zpracovává vstupní sekvenci znaků [13]

## 4.2 Syntaktický strom

Syntaktický strom představuje datovou strukturu, kterou ANTLR vytvoří při parsování vstupního souboru. Struktura je složená z kořene (root) a uzlů, které mohou představovat buď další podstromy, které odpovídají pravidlům gramatiky, nebo listy stromu.

V naší aplikaci Syntaktický strom představuje třída AST 4.2, která uchovává všechna důležitá data pro korektní analýzu vstupního souboru, např. počet uzlů stromu, soubor vztahující se ke konkrétnímu stromu, kořen stromu, atd...

---

```
export class AST {

    static nodeCount: number = 0;
    private parseTree : Node[];
    public documentName: string;
    public currentNode : Node;
    public root : Node;

    constructor(documentName: string) {

        this.parseTree = [];
        this.documentName = documentName;
        this.currentNode = new Node(undefined,undefined,undefined,undefined,
            undefined);
        this.root= new Node(undefined,undefined,undefined,undefined,undefined);

    }

    getParseTree() {
        return this.parseTree;
    }

    addNode(node : Node) : void {
        this.parseTree.push(node);
    }

    findNode(ruleNumber: number) : Node | undefined {

        for(let i = this.parseTree.length-1; i >= 0; i--) {
```

```

        if(this.parseTree[i] != null && this.parseTree[i].getContext()?.
            ruleIndex === ruleNumber) {
            return this.parseTree[i];
        }
    }
    return undefined;
}
}

```

---

Listing 4.1: třída AST v jazyce Typescript

Další, neméně důležitou, třídou v naší aplikaci je třída Node, která představuje uzel stromu. Jedná se, ve své podstatě, o strukturu uchovávající následující atributy:

1. kontext daného uzlu
2. předka uzlu
3. potomka uzlu
4. hodnotu představující text daného uzlu
5. typ uzlu, který se odvíjí od pravidel v gramatice

---

```

export class Node {

    private id : number;
    private context : ParserRuleContext | undefined ;
    private parent : Node | undefined;
    private child : Node[] | undefined;
    private value: string | undefined;
    private _type : number | undefined;

    constructor(context: ParserRuleContext | undefined, parent : Node | undefined,
        child : Node[] | undefined, value : string | undefined, _type: number |
        undefined)

    {
        this.id = AST.nodeCount;
        this.context = context;
        this.parent = parent;
    }
}

```

```

    this.child = child;
    this.value = value;
    this._type = _type;

    AST.nodeCount++;

}

getId() { return this.id; }
getContext() { return this.context; }
getParent() { return this.parent; }
getChildren() { return this.child; }
getValue() {return this.value; }
getType() {return this._type; }
addChild(child : Node) : void{ this.child?.push(child); }

}

```

---

Listing 4.2: třída Node v jazyce Typescript

### 4.3 ANTLR Listener a callback funkce

ANTLR disponuje dvěma mechanismy, které umožňují průchod stromem (tzv. "tree-walking mechanisms"). Jedná se o mechanismy **listener** a **visitor**, přičemž výchozím je listener. Největší rozdíl mezi nimi spočívá v tom, že listener metody jsou volány nezávisle ANTLR objektem, zatímco visitor metody vyvolávají rovněž metody potomků uzlu, na kterém se právě nachází, což způsobuje, že některé podstomy nebudou při průchodu vůbec navštíveny. Samotné listenery jsou ekvivalentní SAX objektům, které se používají v XML parserech.

Aby bylo možné stromem procházet a při průchodu vyvolat odpovídající události, ANTLR disponuje třídou `ParseTreeWalker`. Právě ona se stará o to, aby byly volány callback funkce. Další důležitou komponentou, vygenerovanou ANTLR nástrojem je rozhraní `ParseTreeListener` (v našem případě `MonkeyCListener` 3.2). Toto rozhraní definuje veškeré metody potřebné k kompletnímu průchodu stromem, např. `"enterFunctionDeclaration(context: FunctionDeclarationContext)"` či `"exitFormalParameterDeclarations(context: FormalParameterDeclarationsContext)"` 4.3.

Ke každému pravidlu obsaženému v gramatice, parser vygeneruje 2 metody a kontextovou třídu. První metoda vždy začíná prefixem "enter" a druhá prefixem "exit", kontext poté odpovídá příslušnému pravidlu. Ve chvíli kdy `ParseTreeWalker` narazí na příslušný uzel, vyvolá odpovídající metodu, podle toho zda do uzlu vstupuje, nebo jej opouští.

```

enterFunctionDeclaration(context: FunctionDeclarationContext);
exitFunctionDeclaration(context: FunctionDeclarationContext);
enterFormalParameterDeclarations(context: FormalParameterDeclarationsContext);
exitFormalParameterDeclarations(context: FormalParameterDeclarationsContext);

```

Obrázek 4.3: funkce vygenerované nástrojem ANTLR

## 4.4 Sémantická analýza

Sémantická analýza, ve většině případů, spočívá v nalezení a uložení všech identifikátorů vyskytujících se ve zdrojovém kódu programu. Identifikátory jsou myšleny proměnné, konstanty, funkce, atd... Při analýze jsou dále vyhledávány informace o jednotlivých identifikátorech, např. datový typ proměnné, zda-li již byla deklarována, zda je správně použita vzhledem k jejímu datovému typu, atd... Pro uložení všech těchto informací se, opět ve většině případů, používá tabulka symbolů. Na obrázku 4.4 lze vidět, jak taková tabulka symbolů může vypadat. Tabulka o každé proměnné uchovává její název, velikost v Bytech, zda již byla v programu deklarována a použita.

Tabulka symbolů působí jako velmi efektivní nástroj pro sémantickou analýzu. Dokáže uchovávat všechny identifikátory v kódu a informace o nich. V této práci však tabulka symbolů použita není. Hlavním důvodem byla nedostatečná znalost této problematiky na začátku tvorby práce.

Sémantická analýza je v této práci řešena ve třídě `documentHandler.ts`, která je detailněji popsána v další kapitole. Pro uchovávání informací o identifikátorech v kódu jsou použity Mapy. Typescript Mapa představuje datovou strukturu, která byla nově představena v ECMAScriptu 6 [15]. Mapa dokáže ukládat data ve formátu klíč - hodnota, stejně jako Mapy v ostatních programovacích jazycích, např. Java, C Sharp.

Mapy, které rozšíření používá jsou následující:

1. **diagnosticMap : Map<string, vscode.DiagnosticCollection>** - Mapa, která pro každý dokument uchovává chyby ve zdrojovém kódu programu. Pro detekování těchto chyb je použita třída `ErrorListener.ts` (popsána v kapitole 5).

Adresa	Typ	Název	Délka	Deklarováno	Použito
00000020	double array 10	dbl_ary	80B	A	A
20000004	integer	pom	4B	A	N
20000008	double	i	8B	A	A
2000000c	real	konc	4B	A	N
20000018	integer	indexDoPole	7B	N	A

Obrázek 4.4: ukázka tabulky symbolů [14]

2. **documentAutocompleteMap : Map<string, Map<string, vscode.CompletionList>**  
- Mapa, ve které jsou uloženy všechny proměnné, funkce, konstanty. Tyto data poté rozšíření našeptává podle toho, jaký provider je zrovna spuštěn (pozn. provider jsou popsány v kapitole 5). Klíčem pro tuto mapu je název dokumentu, ke kterému se vztahuje. Hodnotou je opět Mapa, jejíž klíčem je druh identifikátoru (např. **localVariables** - pro lokální proměnná, **functions** - pro funkce) a hodnotou je seznam těchto identifikátorů (tedy např. seznam proměnných, seznam funkcí,...).
3. **abstractSyntaxTreeMap : Map<string, AST | any>** - Mapa, která pro každý dokument uchovává jeho syntaktický strom vygenerovaný ANTLR nástrojem. Hodnotou je název příslušného dokumentu, hodnotou pak instance třídy AST 4.1. Tato mapa je zde zařazena, jelikož vyhledávání v syntaktickém stromě je rovněž součástí sémantické analýzy.
4. **abstractSyntaxTreeCommentaryMap : Map<string, Token[] | any>** - Mapa, ve které jsou uloženy všechny komentáře z daného dokumentu. Nástroj ANTLR při generování syntaktického stromu totiž neukládá samotný kód a jeho komentáře společně. Kód a komentáře jsou rozděleny do kanálů, přičemž kód je uložen v kanálu s indexem 0, a komentáře jsou uloženy v kanálu 1. Bylo proto nutné vytvořit 2 mapy, jednu pro syntaktický strom, a druhou pro komentáře. Význam a využití komentářů v rozšíření je vysvětlena v další kapitole.



## Kapitola 5

# Návrh a implementace rozšíření

Rozšíření bude implementováno v jazyce Typescript. K vytvoření rozšíření samotného je potřeba Node.js [16] a Git [17]. Poté je vyžadována instalace programů Yeoman [18] a VS Code Extension Generator [19], pomocí kterých jsou rozšíření generovány.

### 5.1 Třída `extension.ts`

Základní třída, která je součástí každého VS Code rozšíření, je třída **`extension.ts`**. Na začátku třída obsahuje pouze jedinou funkci, a to *function activate(context: vscode.ExtensionContext)*. Tato funkce rozšíření aktivuje, pokud narazí na soubor s příponou **`.mc`**. Seznam přípon souborů, při jejich otevření se rozšíření aktivuje, je uveden v konfiguračním `package.json` souboru, viz 5.1. Rozšíření se také aktivuje v případě, že pracovní adresář obsahuje Monkey C soubor. V `package.json` souboru jsou mimo jiné uvedeny informace, jako název rozšíření, jeho popis, autor, aktuální verze, atd...

---

```
"activationEvents": [  
  "onLanguage:monkeyc",  
  "workspaceContains:.mc"  
]
```

---

Listing 5.1: aktivační události rozšíření

Funkce ***activate()*** přijímá parametr `context`. Tento parametr reprezentuje nástroje, se kterými rozšíření pracuje. Tělo funkce se skládá z několika podpůrných tříd, tzv. providerů. Providery jsou součástí VS Code API (jedná se o sadu JavaScript rozhraní, které mohou být vyvolány rozšířením) [20].

První provider ve třídě `extension` zajišťuje obarvování Monkey C syntaxe, viz. 5.6. Dále následuje

funkce `onDidOpenTextDocument()`. Jedná se o událost, která se vyvolá při otevření textového dokumentu. Zde je také řešeno parsování souborů. Pokud uživatel otevře pracovní adresář, tedy složku, která obsahuje více souborů, spustí se funkce `parseAllDocuments()`. Událost `onDidChangeTextDocument()` společně s funkcí `parseCurrentDocument()` dále řeší parsování aktuálně upravovaného dokumentu, a to vždy když dojde k nějaké změně. Dále třída `extension.ts` obsahuje sadu providerů, které zajišťují našeptávání a dokončování kódu 5.3. Tento provider přijímá následující tři parametry:

1. **selector** - definuje dokumenty, na které má být provider aplikován
2. **provider** - definuje konkrétní provider
3. **...triggerCharacters** - jedná se o pole znaků, při kterých má být provider aktivován. V rozšíření jsou pro aktivaci providerů nejčastěji použity symboly tečky a mezery

**keywordsProvider** - Provider zajišťující našeptávání klíčových slov jazyka. Pro získání vhodných klíčových slov je použit c3 Engine, popsán v sekci 5.7

**importedModulesProvider** - Pokud jsou ve zdrojovém kódu programu, prostřednictvím klíčového slova **using**, importovány nějaké moduly, tento provider vrátí jejich seznam. V následujícím výpisu kódu je importováno 5 modulů a 1 třída z Toyboxu. Seznam který provider vrátí bude tedy obsahovat 5 názvů modulů a 1 název třídy [WatchUi, Graphics, System, Lang, Object, Time], kde Object je název třídy. V tomto seznamu jsou hodnoty uloženy jako **vscode.CompletionItem** 5.1. Jedná se o třídu, která je součástí VS Code API. Její konstruktor přijímá 2 parametry, název položky a její druh. Druh položky, `CompletionItemKind`, představuje výčet, který obsahuje prvky, jako modul, třída, proměnná, funkce, konstanta, atd...

---

```
using Toybox.WatchUi;
using Toybox.Graphics;
using Toybox.System;
using Toybox.Lang;
using Toybox.Lang.Object;
using Toybox.Time;
```

---

Listing 5.2: importované moduly z Toyboxu

---

```
registerCompletionItemProvider(selector: DocumentSelector, provider:
    CompletionItemProvider, ...triggerCharacters: string[]): Disposable
```

---

Listing 5.3: aktivační události rozšíření

```

constructor CompletionItem(label: string, kind?: vscode.CompletionItemKind | undefined): vscode.CompletionItem
Creates a new completion item.
Completion items must have at least a label which then will be used as insert text
as well as for sorting and filtering.
@param label — The label of the completion.
@param kind — The kind of the completion.

```

Obrázek 5.1: popis konstruktoru třídy CompletionItem z VS Code API

**localVariableProvider** , **classVariableProvider** -Providery zajišťující našeptávání lokálních proměnných ve zdrojovém kódu. V Monkey C existují 2 způsoby, jak lze přistupovat k proměnným. Pokud se jedná o vnitřní proměnnou třídy, přístup k ní není nijak omezen. Pokud je proměnná označena jako **public** nebo **protected**, je potřeba k ní přistoupit pomocí prefixů **self.** nebo **me..**

**functionProvider** - Provider pro funkce ve zdrojovém kódu. Výše bylo zmíněno, že každá funkce musí být před použitím deklarována a nelze ji předat přímo, jako parametr jiné funkci. Pokud však použijeme funkci **method()** z třídy Toybox.Lang.Object, instance této třídy dokáže vytvořit objekt třídy Toybox.Lang.Method, díky které je tuto funkci vyvolat, jako callback, viz. výpis kódu 5.4.

**toyboxProvider** - slouží pro našeptávání komponent Toyboxu při importování do aplikace, např. modulů a tříd, viz. výpis kódu 5.2. Pro získání modulů je použita funkce *findModuleBodyMembers()*, která prochází syntaktickém strom a hledá uzel, ve kterém je uložen Toybox. Jelikož je tento modul rodičem pro všechny ostatní moduly, stačí nalézt první uzel, který obsahuje pravidlo *MonkeyCParser.RULE<sub>m</sub>oduleBodyMembers*, viz. výpis kódu 5.5. Následně stačí pomocí metody **getChildren()** získat všechny jeho potomky. Z těchto potomků poté funkce **collectClassesFromModules()** získá názvy všech tříd, které je možné importovat.

---

```

//! Constructor
function initialize()
{
    View.initialize();
    Sensor.enableSensorEvents( method(:onSnsr) );
}

function onSnsr(sensor_info)
{
    //function body
}

```

---

Listing 5.4: ukázka použití funkce, jako callback

---

```
if(tree[i].getContext()!?.ruleIndex === MonkeyCParser.  
    RULE_moduleBodyMembers) {  
    modules = tree[i].getChildren();  
    break;  
}
```

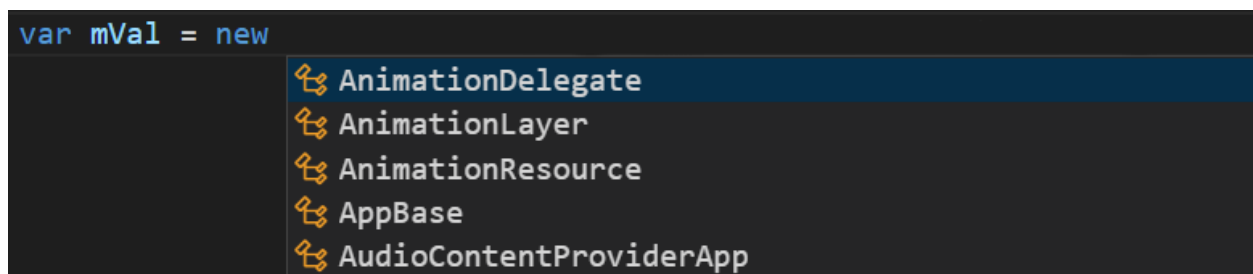
---

Listing 5.5: funkce prochází strom a hledá uzel, ve kterém je uložen Toybox

**accessibleMembersProvider** - Tento provider zajišťuje především našeptávání na proměnných nějakého typu, jak je možné vidět na obrázku 5.6, kde rozšíření našeptává metody z třídy `Toybox.Lang.String`. Pokud je tedy proměnná například instance třídy, nebo se jedná o importovaný modul, provider podle toho v syntaktickém stromě nalezne, co se pod daným identifikátorem nachází, a podle toho jsou volány příslušné funkce. Například pro nalezení všech přístupných členů instance třídy je použita funkce **collectAccessibleMembers()**.

**inheritedMembersProvider** - Jsou deklarovány třídy **A** a **B**, kde třída **B** je potomkem třídy **A**. V Monkey C je dědičnost podporována a k její identifikaci je použito klíčové slovo **extends**. V tomto případě by tedy deklarace třídy B vypadala *class B extends A*. Pokud chceme přistoupit ke členům rodičovské třídy A, provede tento provider příslušné operace a vrátí seznam dostupných členů.

**importedMembersProvider** - Tento provider se stará o našeptávání tříd z importovaných modulů. Na obrázku 5.2 lze vidět, jak rozšíření našeptává prvních 5 tříd z importovaného modulu `Toybox.Application`. Provider je aktivován, pokud se na řádku nachází klíčové slovo **new**, tím pádem pozná, že chce uživatel vytvořit novou instanci třídy.

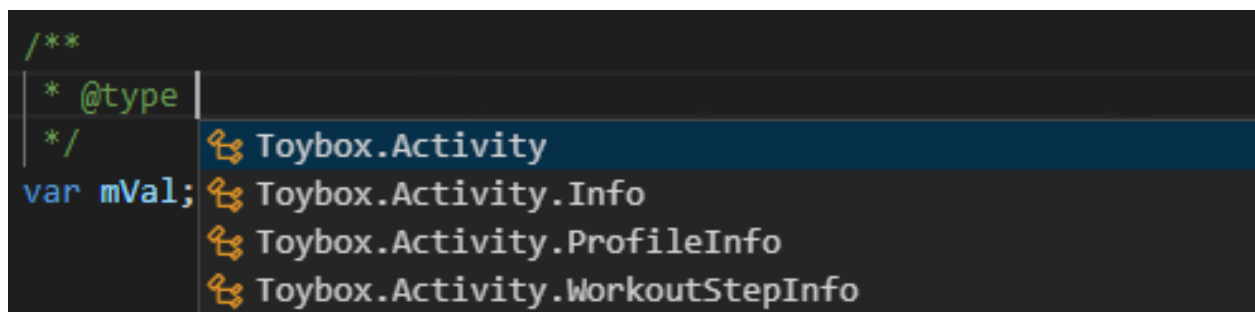


Obrázek 5.2: rozšíření našeptává třídy z importovaných modulů.

**curlyBracesProvider**, **normalBracesProvider** - Zde bylo záměrem automaticky do dokumentu doplnit pravou závorku po zadání levé. Například v situaci, kdy uživatel napíše levou závorku '(', rozšíření automaticky do textu doplní druhou. Doplnit takto závorku do dokumentu se však nepodařilo. Místo toho provider uživateli závorku navrhne, stejně jako navrhuji kandidáty ostatní providery, a ten ji následně po stisknutí tlačítka tab na klávesnici doplní.

**multilineCommentProvider** - Tento provider zajišťuje našeptávání 2 typů víceřádkových komentářů. První typ je klasický prázdný víceřádkový komentář, který nalezneme ve většině programovacích jazyků. Druhý typ, který je možné vidět na obrázku 5.7, je specifický tím, že v sobě obsahuje **@type**. Tento komentář slouží k popisu proměnné při její deklaraci. Za prefix **@type** uživatel uvede datový typ proměnné. Je zde využito toho, že v Monkey C musí být proměnná před použitím deklarována, jak už zde bylo několikrát zmíněno a jak je popsáno v sekci 5.7.

**dataTypesProvider** - Další a zároveň poslední provider ve třídě **extension.ts**, který úzce souvisí s providerem předchozím, je **dataTypesProvider**. Jak už je z názvu patrné, provider zajišťuje našeptávání všech datových typů z Toyboxu. Na obrázku 5.3 lze vidět, jak rozšíření našeptává datové typy z Toyboxu.



Obrázek 5.3: rozšíření našeptává datové typy z modulu Toybox.Activity.

Na začátku této sekce byla zmíněna funkce *activate()*, která je volána při spuštění rozšíření. K ní je, jako protiklad, na konci funkce *deactivate()*, která je volána v momentě, kdy je deaktivováno rozšíření.

## 5.2 Třída **documentHandler.ts**

Jedná se o jednu z nejdůležitějších a nejrozsáhlejších tříd v celé aplikaci. Obsahuje přes 30 metod, které zajišťují plynulý chod rozšíření. Nacházejí se zde Mapy, ve kterých jsou uloženy veškeré informace jednotlivých dokumentů, jak je popsáno v sekci 4.4. Nejdůležitějšími třídami jsou však **parse-**

```

inputStream = new ANTLRInputStream(readFileSync(fileUri.fsPath, 'utf-8'));
lexer = new MonkeyCLexer(inputStream);
lexer.removeErrorListeners();
lexer.addErrorListener(this.errorListener);
tokenStream = new CommonTokenStream(lexer);
this.parser = new MonkeyCParser(tokenStream);
this.parser.buildParseTree = true;
this.parser.removeErrorListeners();
this.parser.addErrorListener(this.errorListener);
parseTree = this.parser.program();
let shifter = new CommentShifter(tokenStream);
//parse tree for source code
ParseTreeWalker.DEFAULT.walk(listener, parseTree);
//parse tree for comments
ParseTreeWalker.DEFAULT.walk(shifter, parseTree);

```

Obrázek 5.4: vytvoření instancí parseru a lexeru a následné parsování souboru.

*AllDocuments()* a *parseCurrentDocument()*. Funkce *parseAllDocuments()* je při aktivaci rozšíření volána jako první. Zde dochází k načtení všech souborů z pracovního adresáře. Následně je pro daný soubor vytvořen *inputStream*, jenž je jako parametr předán instanci **MonkeyCLexer**u. Dále dojde k vytvoření *tokenStream*u, jenž je předán instanci **MonkeyCParser**u. Poté dojde k parsování dokumentu a vytvoření syntaktického stromu jak pro zdrojový kód, tak pro komentáře, viz. obrázek 5.4.

Dále je volána funkce **provideAutocomplete()**, která ze syntaktického stromu získá proměnné, funkce, atd... Vždy pro konkrétní dokument. Funkce *updateCollection()* poté zpracuje všechny syntaktické chyby, která detekoval Error Listener 5.4.

Funkce *parseCurrentDocument()* vykonává stejnou činnost, jako výše popsaná pouze s tím rozdílem, že tato funkce je volána vždy při práci s konkrétním dokumentem.

### 5.3 Třída Listener.ts

Jedná se o třídu, která implentuje rozhraní **MonkeyCListener**. Toto rozhraní definuje kompletní listener pro parser vytvořený **MonkeyCParser**em. jinými slovy, nacházejí se zde všechny metody, které jsou potřeba pro vytvoření syntaktického stromu při průchodu parseru. Příklad vytvořených metod je možné vidět na obrázku 4.3. Kromě zpracovávání zdrojového kódu třída Listener zajišťuje také zpracování komentářů. Toto je zajištěno prostřednictvím třídy *CommentShifter*, převzaté z

Definitive ANTLR 4 reference [21]. Tato třída umožňuje přístup ke skrytému kanálu s komentáři, které jsou od zdrojového kódu odděleny.

## 5.4 Error Listener

Třída, která slouží k detekování zachytávání syntaktických chyb v kódu. Jedná se o chyby, které jsou v rozporu z pravidly gramatiky. Každá chyba, kterou rozšíření detekuje, je uložena do struktury **ErrorDescription** 5.6. Všechny chyby jsou poté uloženy v poli a pomocí metody `getSyntaxErrors()` je možné je získat. Rozšíření chyby vypisuje do konzole, jak je ve VS Code zvykem. Součástí zprávy jsou:

1. řádek, na kterém se chyba nachází
2. pozice na řádku
3. popis chyby

---

```
export interface ErrorDescription {  
    document: string;  
    offendingSymbol: any;  
    line: number;  
    charPositionInLine: number;  
    msg : string  
    e: RecognitionException | undefined  
}
```

---

Listing 5.6: rozhraní pro popis chyby

## 5.5 Modul Toybox

Modul Toybox představuje v Monkey C jmenný prostor (namespace), pod kterým jsou seskupovat třídy, metody, funkce, atd... Obsahuje všechny potřebné třídy, které Monkey C poskytuje, na jednom místě. Z názvů jednotlivých modulů a tříd lze jednoduše odvodit, jaké metody se zde nachází a k jakému účelu slouží.

Jelikož tento modul, není nikde oficiálně dostupný. Respektive neexistuje žádná oficiální verze, kterou by bylo možné použít, bylo nutné sestavit modul vlastními silami. K jeho vytvoření bylo použito oficiální Connect IQ SDK. V tomto SDK jsou obsaženy informace, ke všem komponentům Toyboxu. Problémem bylo, že zdrojem těchto informací byly soubory ve formátu .html, tedy webové stránky. Bylo tedy nutné veškeré informace extrahovat z html elementů a následně je sestavit do požadované

```

1  using Toybox.WatchUi;
2  using Toybox.System;
3
4  class A {
5
6      private var privateVal;
7
8      // Update the view
9      function onUpdate(dc) {
10
11          // Get and show the current time
12          var clockTime = System.getClockTime();
13
14      }
15  }
16

```

(a) před

```

1  using Toybox.WatchUi;
2  using Toybox.System;
3
4  class A {
5
6      private var privateVal;
7
8      // Update the view
9      function onUpdate(dc) {
10
11          // Get and show the current time
12          var clockTime = System.getClockTime();
13
14      }
15  }

```

(b) po

Obrázek 5.5: Monkey C kód před a po přidání obarvení syntaxe

formy. Jelikož bylo vygenerování Toyboxu velmi pracné a extrakce jednotlivých částí z html elementů vyžadovala mnoho úsilí, není zaručeno, že je Toybox zcela kompletní. Vygenerovaný Toybox například neobsahuje deklarované konstanty, které se v některých třídách v oficiální dokumentaci nachází. Velkou výhodou by bylo, kdyby společnost Garmin Ltd. [4] vydala svoji plnohodnotnou verzi Toyboxu. Toto by při vývoji rozšíření ušetřilo velké množství práce a času stráveném při jeho generování.

## 5.6 Obarvení kódu

Zvýraznění a obarvení Monkey C syntaxe bylo převzato z GitHub repozitáře [22]. Autor Alexander Fedora [23] zde k obarvení klíčových slov Monkey C využívá JSON souboru. Tento soubor obsahuje všechna klíčová slova, ty pomocí regulárních výrazů vyhledává v textu a poté je obarvuje.

Hned na první pohled je zřejmé, že obarvení poskytuje uživateli větší přehled a orientaci v kódu, jak je vidět na obrázku 5.5b.

## 5.7 Automatické doplňování a našeptávání kódu

Jako první bylo v rozšíření řešeno našeptávání klíčových slov jazyka, např. NEW, VAR, FUNCTION, atd... k tomuto účelu byl použit "The ANTLR4 Code Completion Core"[24]. Jedná se o "stroj"sloužící k dokončování kódu pro analyzátoři založené na ANTLR4. Engine c3 je schopen poskytnout kandidáty na doplnění kódu, kteří jsou užiteční pro editory s analyzátoři generovanými ANTLR, nezávisle na skutečném jazyku / gramatice použité pro generování. Původní implementace je poskytována v jazyce Typescript, což bylo vhodné použít vzhledem k tomu, že rozšíření je také psáno v Typescriptu.

Dále bylo řešeno našeptávání lokálních funkcí a proměnných. V Monkey C jsou k tomuto účelu



použity 2 prefixy **self.** a **me..** Pokud uživatel v kódu zadá jeden z těchto prefixů, rozšíření spustí funkci **provideAutocomplete**, která pomocí průchodu syntaktickým stromem všechny dostupné proměnné, funkce, či třídy, podle toho, kde se zrovna uživatel v kódu nachází.

V neposlední řadě bylo potřeba vyřešit, jak zjistit, co se nachází v konkrétní proměnné, a na základě toho poté provést příslušné našeptávání. Toto je řešeno pomocí komentářů. Tyto Komentáře se objevují jednak v Toyboxu, kde slouží pro orientaci v už tak rozsáhlém modulu a popisu jednotlivých jeho součástí. Součástí popisu jsou informace o datových typech, vstupních parametrech (pokud se jedná o funkci), návratových typech atd...

Komentáře má také k dispozici uživatel přímo v kódu. Každá proměnná, která je deklarována, by měla na sebou obsahovat komentář nesoucí informaci o datovém typu této proměnné. Komentář má jednoduchou syntax, viz. 5.7, a rozšíření je navíc schopné jeho strukturu automaticky doplnit po zadání `"/**`. Je zde využito toho, že každá proměnná v Monkey C musí být deklarována předtím, než ji lze použít. A právě na základě tohoto byly v rozšíření komentáře implementovány. Není tedy potřeba složitě hledat a ukládat informace o datovém typu do nějaké struktury, stačí pouze v kanálu komentářů najít příslušný řádek, na kterém je proměnná deklarována a z něj datový typ extrahovat.

---

```
/**
 * @type Toybox.Lang.Number
 */
```

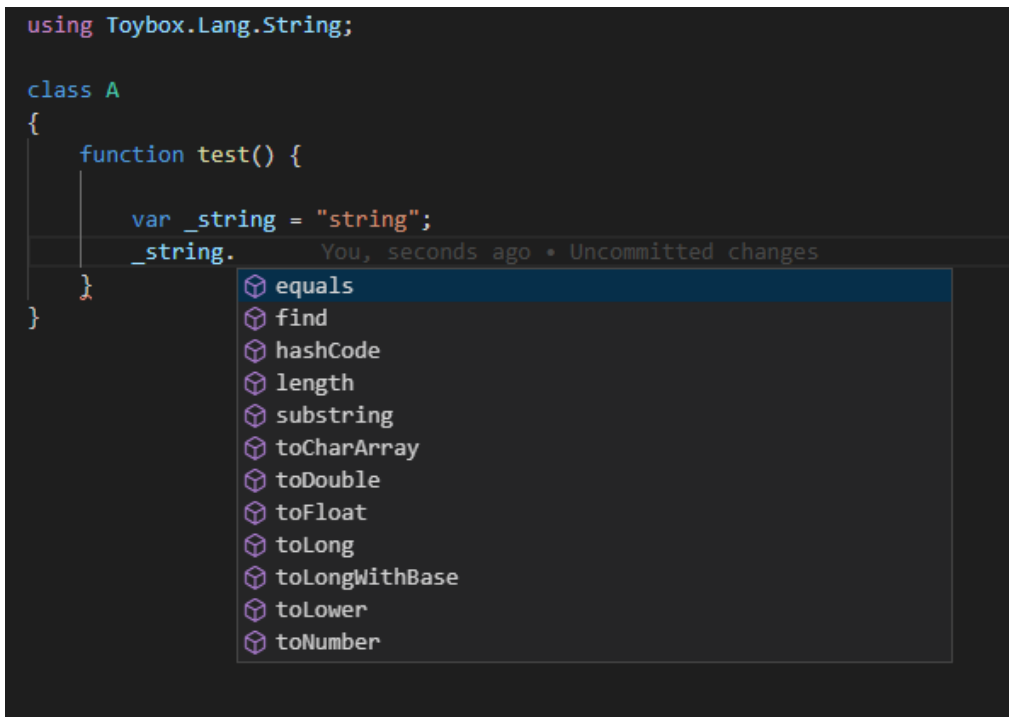
---

Listing 5.7: struktura komentáře datový typ

## 5.8 Popis funkcí a proměnných Toyboxu

Rozšíření dokáže při našeptávání jednotlivých částí Toyboxu zobrazit jejich název, druh a popis. A právě k jejich popisu bylo využito komentářů, která každá funkce a proměnná obsahuje. Tyto komentáře vznikly při generování Toyboxu a kromě popisu mají mnoho dalších účelů popsanych výše (např. našeptávání datového typu proměnné). Na obrázku 5.8 lze vidět popis funkce `registerSensorDataListener()` z modulu `Toybox.Sensor`. Součástí popisu je:

1. návratový typ funkce
2. název funkce
3. popis funkce
4. jednotlivé parametry společně s jejich datovým typem



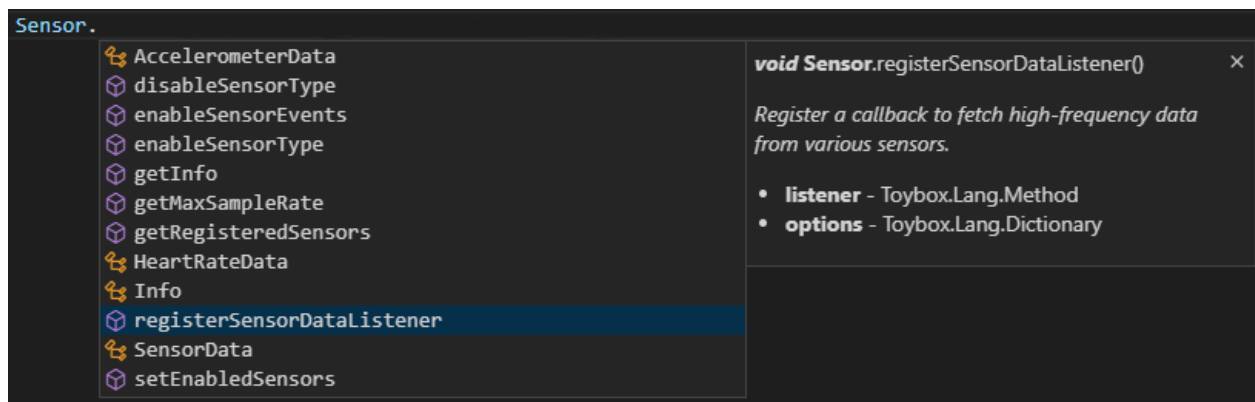
Obrázek 5.6: ukázka automatického našeptávání kódu na proměnné typy string

## 5.9 Nedostatky rozšíření

Při implementaci automatického našeptávání byl detekován problém, kvůli kterému není možné provést volání více funkcí po sobě na jednom řádku. Uvedme si příklad, kdy máme proměnnou typu **String**, v níž je uloženo číslo. Hodnotu v této proměnné budeme chtít převést na typ Integer, čili zavolat metodu `toNumber()`, a poté bezprostředně po volání `toNumber()` zavolat další metodu. Zde nastává problém, kdy rozšíření, jako další vstup neočekává možné volání funkce 5.10. Jádrem tohoto problému je, že poskytnutá bezkontextová gramatika popisující jazyk není stoprocentně přesná. A právě kvůli těmto "nepřesnostem" je možné při implementaci narazit na podobné komplikace. V

```
public class HeartRateIterator {  
  
    /**  
     * Get the maximum heart rate contained in this iterator.  
     * @param  
     * @returns Toybox::Lang::Number  
     */  
    public function getMax() { }
```

Obrázek 5.7: komentář nad funkcí obsahující stručný popis, parametry funkce a návratový typ



Obrázek 5.8: popis funkce registerSensorDataListener() z modulu Toybox.Sensor.

průběhu vývoje zatím nebyly registrovány další problémy způsobené gramatikou.

```
var _string = "123";    var test = _string.toNumber();
```

Obrázek 5.9: nedostatek rozšíření

```
line 7:61 mismatched input '.' expecting {';', '?', 'instanceof', 'has', 'and', 'or', '[', ',', '*', '|', '<', '>', '&', '||', '&&', '++', '--', '=', '!=', '!=', '!=', '+=', '-=', '*=', '/=', '&=', '|=', '%=', '^', '%', '+', '-', '/'}
```

Obrázek 5.10: chybová hláška z error listeneru

## Kapitola 6

# Testování výsledného řešení

Testování probíhalo pravidelně v průběhu vývoje rozšíření na jednoduchých fragmentech kódu Monkey C jazyka. Pro finální testování však byly použity programy, jenž byly součástí reálných aplikací. Tyto příkladové aplikace byly převzaty z oficiální Connect IQ SKD.

První testovací zdrojový kód je možné vidět na výpisu **A.1**. Tento kód komunikuje se senzory obsažených v Garmin zařízení a s jejich pomocí měří srdeční tep uživatele. Při testování tohoto kódu nedošlo k žádným výraznějším problémům. Při deklaraci jednotlivých tříd, které vždy byly rozšířeny jinou třídou pomocí **extends**, rozšíření správně našeptávalo možné kandidáty na doplnění. Dále ve funkci *initialize()*, kde je volána metoda *enableSensorEvents()*, bylo rozšíření schopno našeptávat callback funkce po zadání znaku ':'. Některé části kódu bylo však nutné upravit. Jako první bylo nutné přidat komentáře s datovým typem nad deklarace proměnných *stringHR* a *HRgraph*. Zde však nastal problém, kdy nebylo možné jednoznačně určit, jaký datový typ má proměnná *HRgraph* obsahovat. Ve funkci *initialize()* je do této proměnné vložena instance třídy **LineGraph**. Tuto třídu však rozšíření nenašlo a po zkontrolování oficiální Monkey C API [25] dokumentace bylo zjištěno, že takovou třídu API neobsahuje, tudíž ji rozšíření ani nemohlo nabídnout, jako kandidáta na doplnění.

Druhý testovací zdrojový kód je možné vidět na výpisu **A.2**. Jedná se o část aplikace *GenericChannelBurst*, která pracuje s třídami *Toybox.Ant.BurstListener*, *Toybox.Ant.Ant.GenericChannel*, *Toybox.Ant.ChannelAssignment*, atd... U toho zdrojového kódu byl kladen důraz především na práci s konstantami. Hned před deklarací třídy *BurstChannel* lze vidět první konstantu **ANT-DATA-PACKET-SIZE**. Další konstanty následují již v těle třídy. Na obrázku 6.1 lze vidět, jak rozšíření našeptává konstanty obsahující písmeno **D**. Lze si také všimnout, že i lokální proměnné deklarované uživatelem obsahují vlastní popis. V tomto případě je součástí popisu datový typ proměnné. Při importování modulů pomocí **using**, deklaraci třídy společně s použitím dědičnosti a deklaraci všech proměnných nebyl detekován žádný problém. Dále byla testována deklarace nového pole, práce s **for** cyklem, volání metod na proměnných atd...

```
const DEVICE_NUMBER = 123;  
const DEVICE_TYPE = 1;  
const FREQUENCY = 66;  
const PERIOD_1_HZ = 32768;  
const TRANS_TYPE = 0;  
  
hidden var _transmissionCounter;
```

D

DEVICE_NUMBER	const variable
DEVICE_TYPE	
do	
ANT_DATA_PACKET_SIZE	

Obrázek 6.1: rozšíření našeptává konstanty obsahující písmeno D.

# Kapitola 7

## Závěr

Cílem této bakalářské práce bylo nastudovat jazyk Monkey C, nástroj ANTLR, díky kterému jsme schopni analyzovat a parsovat formální jazyky, a s jeho pomocí vytvořit parser jazyka Monkey C, jenž byl následně použit k analyzování kódu.

V úvodní části této práce se hovoří o tom, jak znatelně malé zastoupení má jazyk Monkey C na trhu s rozšířeními pro VS Code, vzhledem k ostatním programovacím jazykům. Tato skutečnost byla také jeden z hlavních důvodů, které vedly k tvorbě vlastního řešení. Dále byl v kapitole 2 představen a popsán jazyk Monkey C, který je určen k vývoji aplikací a rozšíření pro zařízení Garmin, byly popsány druhy aplikací, které je možné v Monkey C vyvíjet. Závěrem této kapitoly bylo porovnání Monkey C s ostatními moderními programovacími jazyky. Při vývoji rozšíření a následném testování jsem došel k závěru, že jazyk Monkey C má s ostatními, jako Python či Java mnoho společného.

Hlavním výsledkem práce je rozšíření pro VS Code, které uživateli napomáhá s psaním Monkey C kódu. Návrhem a implementací se zabývala kapitola 5. Byly zde popsány všechny důležité komponenty, jako providery pro našeptávání kódu, třída *extension.ts*, ve které jsou providery implementovány, dále třída *documentHandler.ts*, která obsahuje Mapy pro sémantickou analýzu a další podpůrné funkce pro parsování dokumentů, komunikaci s třídou *Listener.ts*, která obsahuje callback funkce, jenž odpovídají pravidlům gramatiky popsané v kapitole 4, atd... Během vývoje byly řešeny problémy, jako jsou detekce modulů, funkcí, tříd a proměnných. Bylo řešeno našeptávání proměnných a funkcí, které jsou obsaženy ve třídě, pomocí klíčových slov **self**. a **me.**, které Monkey C využívá. Bylo řešeno parsování všech souborů, pokud se jich v pracovní složce nachází více. Dále byla řešena viditelnost mezi soubory, obarvení kódu pro jeho zpřehlednění, výpis chyb prostřednictvím *ErrorListeneru*, detekování datového typu proměnné pomocí komentáře, jenž je její součástí, atd... Součástí práce je i vygenerovaný modul Toybox, obsahující všechny potřebné třídy a funkce. Rozšíření bylo testováno na ukázkových kódech z oficiálního Connect IQ SDK.

V této práci jsem uplatnil mnoho znalostí napříč různými jazyky. Využil jsem znalost jazyka JavaScript, jenž má prakticky totožnou syntax s Typescriptem. Dále jsem využil znalosti objektově orientovaného programování, které byly velmi užitečné při vývoji všech podpůrných tříd použitých

ve finální aplikaci.

Díky této práci jsem měl také možnost rozšířit své znalosti o práci s jazykem Typescript. Dále jsem měl možnost pracovat s nástrojem ANTLR, který mě umožnil nahlédnout do problematiky překladačů a vývoji jazykových procesorů. Po dobu tvorby práce byl veškerý postup zaznamenáván pomocí verzovacího nástroje Git.

Do budoucna by bylo možné třídu `DocumentHandler` rozšířit o tabulku symbolů, která by více zefektivnila sémantickou analýzu kódu, a tím pádem zvýšila účinnost a rychlost rozšíření. Dále by bylo vhodné nalézt efektivnější řešení pro generování Toybox modulu, jenž byl pro tuto práci vygenerován z html elementů obsažených v SDK. Jedním z možných řešení by bylo, kdyby společnost Garmin Ltd. [4] tento modul v nějaké ucelené podobě zveřejnila.

# Literatura

1. *Essential tools for software developers and teams*. JetBrains s.r.o., [b.r.]. Dostupné také z: <https://www.jetbrains.com/>.
2. *Visual Studio Marketplace*. Microsoft Corporation, [b.r.]. Dostupné také z: <https://marketplace.visualstudio.com/vscode>.
3. *Hello Monkey C!* Garmin LTD or Its Subsidiaries, 2021. Dostupné také z: <https://developer.garmin.com/connect-iq/monkey-c/>.
4. *Garmin International: Home*. Garmin Ltd., [b.r.]. Dostupné také z: <https://www.garmin.com/en-US/>.
5. *Garmin Connect IQ: An in-depth introduction to the platform you can now use today*. 2015-01. Dostupné také z: <https://www.dcrainmaker.com/2015/01/connect-iq-intro.html>.
6. VĚNSEK, Tomáš. *Konfigurovatelná aplikace hodinek pro platformu Garmin Connect IQ*. 2019.
7. ABELSON, Harold; SUSSMAN, Gerald Jay. *Structure and interpretation of computer programs*. The MIT Press, 1996.
8. *Visual Studio Code*. Wikimedia Foundation, 2020-06. Dostupné také z: [https://cs.wikipedia.org/wiki/Visual\\_Studio\\_Code](https://cs.wikipedia.org/wiki/Visual_Studio_Code).
9. *TypeScript*. Wikimedia Foundation, 2020-08. Dostupné také z: <https://cs.wikipedia.org/wiki/TypeScript>.
10. PARR, Terence. 2021. Dostupné také z: <https://www.antlr.org/>.
11. *Interpret (software)*. Wikimedia Foundation, 2020-06. Dostupné také z: [https://cs.wikipedia.org/wiki/Interpret\\_\(software\)](https://cs.wikipedia.org/wiki/Interpret_(software)).
12. PARR, Terence. In: *Definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013, s. 20.
13. PARR, Terence. In: *Definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013, s. 10.
14. *Tabulka symbolů*. Wikimedia Foundation, 2019-04. Dostupné také z: [https://cs.wikipedia.org/wiki/Tabulka\\_symbol%C5%AF](https://cs.wikipedia.org/wiki/Tabulka_symbol%C5%AF).
15. *ECMAScript® 2015 Language Specification*. ECMA International, 2019-04. Dostupné také z: <https://262.ecma-international.org/6.0/>.



16. NODE.JS. [B.r.]. Dostupné také z: <https://nodejs.org/en/>.
17. [B.r.]. Dostupné také z: <https://git-scm.com/>.
18. *The web's scaffolding tool for modern webapps*. [B.r.]. Dostupné také z: <https://yeoman.io/>.
19. *Yo Code - Extension and Customization Generator*. [B.r.]. Dostupné také z: <https://www.npmjs.com/package/generator-code>.
20. *VS Code API*. Microsoft, 2016-04. Dostupné také z: <https://code.visualstudio.com/api/references/vscode-api>.
21. PARR, Terence. In: *Definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
22. FEDORA, Alexander. *ghisguth/vscode-monkey-c*. [B.r.]. Dostupné také z: <https://github.com/ghisguth/vscode-monkey-c>.
23. FEDORA, Alexander. *Alexander Fedora*. [B.r.]. Dostupné také z: <https://github.com/ghisguth>.
24. MIKE-LISCHKE. *mike-lischke/antlr4-c3*. [B.r.]. Dostupné také z: <https://github.com/mike-lischke/antlr4-c3>.
25. *API Documentation*. Garmin Ltd., [b.r.]. Dostupné také z: <https://developer.garmin.com/connect-iq/api-docs/>.

## Příloha A

# Testovací zdrojové kódy

---

```
//!  
//! Copyright 2015 by Garmin Ltd. or its subsidiaries.  
//! Subject to Garmin SDK License Agreement and Wearables  
//! Application Developer Agreement.  
//!  
  
using Toybox.WatchUi;  
using Toybox.Graphics;  
using Toybox.System;  
using Toybox.Lang;  
using Toybox.Time.Gregorian;  
using Toybox.Sensor;  
using Toybox.Application;  
using Toybox.Position;  
  
class SensorTester extends WatchUi.View  
{  
    var string_HR;  
    var HR_graph;  
  
    //! Constructor  
    function initialize()  
    {  
        View.initialize();  
        Sensor.setEnabledSensors( [Sensor.SENSOR_HEARTRATE] );  
        Sensor.enableSensorEvents( method(:onSnsr) );  
    }  
}
```

```

    HR_graph = new LineGraph( 20, 10, Graphics.COLOR_RED );

    string_HR = "---bpm";
}

//! Handle the update event
function onUpdate(dc)
{
    dc.setColor(Graphics.COLOR_BLACK, Graphics.COLOR_BLACK);
    dc.clear();

    dc.setColor(Graphics.COLOR_WHITE, Graphics.COLOR_TRANSPARENT );

    dc.drawText(dc.getWidth() / 2, 90, Graphics.FONT_LARGE, string_HR,
        Graphics.TEXT_JUSTIFY_CENTER);

    HR_graph.draw(dc, [0, 0], [dc.getWidth(), dc.getHeight()]);
}

function onSnsr(sensor_info)
{
    var HR = sensor_info.heartRate;
    var bucket;
    if( sensor_info.heartRate != null )
    {
        string_HR = HR.toString() + "bpm";

        //Add value to graph
        HR_graph.addItem(HR);
    }
    else
    {
        string_HR = "---bpm";
    }

    WatchUi.requestUpdate();
}
}

```

```

//! main is the primary start point for a Monkeybrains application
class SensorTest extends Application.AppBase
{
    function initialize() {
        AppBase.initialize();
    }

    function onStart(state)
    {
        return false;
    }

    function getInitialView()
    {
        return [new SensorTester()];
    }

    function onStop(state)
    {
        return false;
    }
}

```

---

Listing A.1: Testovací zdrojový kód 1

---

```

//
// Copyright 2016 by Garmin Ltd. or its subsidiaries.
// Subject to Garmin SDK License Agreement and Wearables
// Application Developer Agreement.
//

using Toybox.Ant;
using Toybox.System;

const ANT_DATA_PACKET_SIZE = 8;

class BurstChannel extends Ant.GenericChannel {
    const DEVICE_NUMBER = 123;
}

```

```

const DEVICE_TYPE = 1;
const FREQUENCY = 66;
const PERIOD_1_HZ = 32768;
const TRANS_TYPE = 0;

hidden var _transmissionCounter;

///! Constructor.
///! Initializes the channel object, sets the burst listener and opens the
channel
///! @param [Number] channelType See Ant.CHANNEL_TYPE_XXX
///! @param [TestBurstListener] listener The BurstListener to assign
function initialize(channelType, listener) {
    // Get the channel
    var chanAssign = new Ant.ChannelAssignment(
        channelType,
        Ant.NETWORK_PUBLIC );
    GenericChannel.initialize(method(:onMessage), chanAssign);

    // Set the configuration
    var deviceCfg = new Ant.DeviceConfig( {
        :deviceNumber => DEVICE_NUMBER,
        :deviceType => DEVICE_TYPE,
        :transmissionType => TRANS_TYPE,
        :messagePeriod => PERIOD_1_HZ,
        :radioFrequency => FREQUENCY } );
    GenericChannel.setDeviceConfig( deviceCfg );

    // Set the listener for burst messages
    GenericChannel.setBurstListener(listener);

    // Open the channel
    GenericChannel.open();

    // Reset the transmission counter
    _transmissionCounter = 0;
}

```

```

    ///! Ant.Message handler
    ///! @param [Message] msg The Message received over the channel
    function onMessage(msg) {
        var payload = msg.getPayload();
        if(Ant.MSG_ID_CHANNEL_RESPONSE_EVENT == msg.messageId)
        {
            if(Ant.MSG_ID_RF_EVENT == payload[0])
            {
                var eventCode = payload[1];
                if(Ant.MSG_CODE_EVENT_TX == eventCode)
                {
                    ///Create and populate the data payload
                    var data = new [ANT_DATA_PACKET_SIZE];
                    for(var i = 0; i < ANT_DATA_PACKET_SIZE; i++)
                    {
                        data[i] = _transmissionCounter;
                    }
                    _transmissionCounter++;

                    ///Form the message
                    var message = new Ant.Message();
                    message.setPayload(data);

                    /// Set the broadcast buffer
                    GenericChannel.sendBroadcast(message);
                }
                else if(Ant.MSG_CODE_EVENT_CHANNEL_CLOSED == eventCode)
                {
                    /// Reopen the channel if it closed due to search timeout
                    GenericChannel.open();
                }
            }
        }
    }
}

```

---

Listing A.2: Testovací zdrojový kód 2